TEMA 6: ANÁLISIS SINTÁCTICO DESCENDENTE.

La idea general de este tema es generar un análisis a izquierdas para una cadena de entrada dada, es decir, encontrar la derivación por la izquierda.

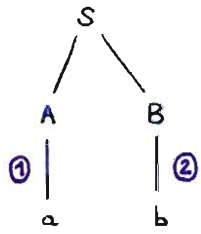
> El árbol se obtiene expandiendo el símbolo no terminal por la izquierda, es decir, el árbol se genera en pre-orden ⇒ Se estudia un nodo y luego su subárbol izquierdo completo. Posteriormente, el subárbol derecho.

Ejemplo: Tenemos las reglas:

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b$



Hay que derivar primero A y hasta que no acabe con toda su rama, no se deriva el símbolo no terminal B.

* ANÁLISIS SINTÁCTICO DESCENDENTE RECURSIVO:

Se basa en una serie de procedimientos a los que se va llamando de manera recursiva para realizar el análisis de la cadena de entrada. Cada uno de estos procedimientos corresponde a un símbolo no terminal de la gramática. Los procedimientos se irán llamando entre sí y el orden de estas llamadas determina el análisis de la cadena.

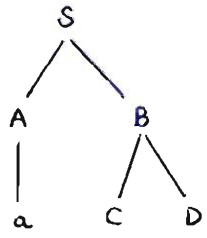
Ejemplo: Gramática:

$S \rightarrow AB$

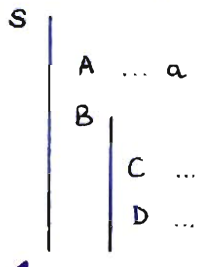
$A \rightarrow a$

$B \rightarrow CD$

Árbol sintáctico:



Traza de la ejecución de procs:



La traza define el árbol sintáctico.

> Los procedimientos recursivos simulan la elaboración del árbol.

1. ANÁLISIS DESCENDENTE RECURSIVO PREDICTIVO.

Es un tipo de análisis descendente recursivo. La característica de este análisis es que, sabiendo el siguiente símbolo de la cadena de entrada, se puede determinar sin ambigüedad el camino a seguir en el análisis sintáctico descendente. Esto quiere decir que, dado un símbolo no terminal con diferentes producciones, se sabe cuál de ellas hay que aplicar.

Este análisis no tiene retroceso, por lo que hay que acertar a la primera en la elección de la producción.

¿Qué problemas me puedo encontrar para la elección de la producción a aplicar?

* Producciones que empiecen de la misma forma:

$$\left.\begin{array}{l} B \rightarrow cCD \\ B \rightarrow cDF \end{array}\right\} \text{¿Cuál de las dos escojo?}$$

* Reglas recursivas por la izquierda ⇒ Generan bucles infinitos.

$$B \rightarrow Ba$$

* REGLAS DE LA GRAMÁTICA:

Cualquier gramática debe cumplir las siguientes reglas para que se pueda utilizar en un análisis sintáctico descendente:

1.- No recursiva por la izquierda.

2.- Las reglas tienen que estar factorizadas por la izquierda (así evitamos tener más de una regla que empiece de la misma forma).

Condiciones necesarias pero no suficientes.

Ejemplo:

$$\left.\begin{array}{l} B \rightarrow cCD \\ B \rightarrow cDF \end{array}\right\} \rightarrow \text{Factorizar generando un nuevo símbolo no terminal.} \longrightarrow \left\{\begin{array}{l} B \rightarrow cT \\ T \rightarrow CD \\ T \rightarrow DF \end{array}\right.$$

1.1. DIAGRAMAS DE TRANSICIÓN.

Organigrama del procedimiento.

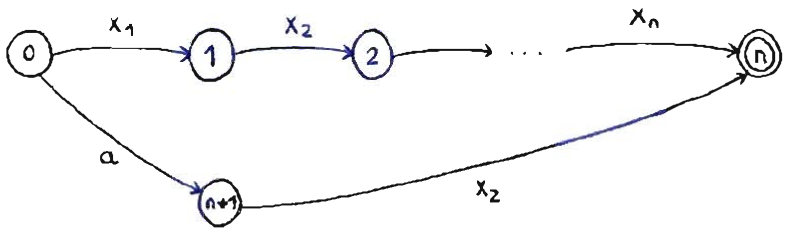
Se trata de una técnica auxiliar para la construcción de procedimientos. Para cada símbolo no terminal de la gramática se construye un diagrama de transición formado por nodos y arcos. Los nodos representan estados y los arcos irán etiquetados con un símbolo terminal, no terminal o λ.

El estado inicial del diagrama corresponde a la llamada del procedimiento, a la entrada del mismo. El estado final indica la salida del procedimiento, la devolución del control al procedimiento que lo había llamado.

Dada la producción $A \rightarrow X_1 X_2 \dots X_n$, en el diagrama de transición de A me tengo que asegurar de que hay una sucesión de estados (nodos) cuyos arcos están etiquetados con la parte derecha de la producción.

Ejemplo:

$$\left.\begin{array}{l} A \rightarrow X_1 X_2 \dots X_n \\ A \rightarrow a\, X_2 \end{array}\right\}$$



Un ARCO etiquetado con un símbolo NO TERMINAL significa ejecutar el procedimiento correspondiente a dicho símbolo no terminal (es una llamada al procedimiento).

≈ token

Un ARCO etiquetado con un símbolo TERMINAL indica que hay que realizar las siguientes acciones:

- Comprobar que es ese terminal el que tengo en la posición actual de la cadena de entrada.
- Validar el terminal (consumir el token).
- Leer el siguiente token (pasar al siguiente elemento).

Ejemplo: Dada la gramática:

$$E \rightarrow E+T \mid T \qquad 1 \mid 2$$
$$T \rightarrow T*F \mid F \qquad 3 \mid 4$$
$$F \rightarrow (E) \mid id \qquad 5 \mid 6$$

Construir los diagramas de transición.

* ¿La gramática cumple las reglas necesarias?

Esta gramática tiene 2 reglas recursivas por la izquierda (1 y 3). Hay que modificarla para poder usar el análisis sintáctico descendente predictivo.

¿Cómo elimino la recursividad por la izquierda?

① $E \rightarrow E+T \longrightarrow E \rightarrow (E)+(T)$

$(E+T)+(T)$

$E+T+T+T \ \ldots$

Para salir del bucle tengo que derivar $E \rightarrow T$. Así, cualquiera de las producciones se convierte en $T+T+T\ldots$

Cambio la recursividad a la derecha:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \lambda$$

} Primero salgo con la T y luego hago la recursividad. Para pararla uso λ.

② $T \rightarrow T*F \longrightarrow T \rightarrow (T)*(F)$

$(T*F)*(F)$

$T*F*F \ \ldots$

Para salir del bucle tengo que derivar $T \rightarrow F$. Así, cualquiera de las producciones se convierte en $F*F*F\ldots$

Cambio igual que en el caso anterior:

$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \lambda$$

}

La nueva gramática es:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \lambda$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \lambda$
$F \rightarrow (E) \mid id$

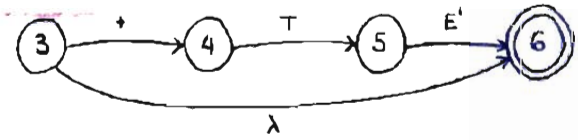
* Ahora sí puedo hacer los diagramas de transición (uno por cada no terminal).

E:

⇒

```
PROCEDURE E;
BEGIN
    T;
    E';
END
```

E':

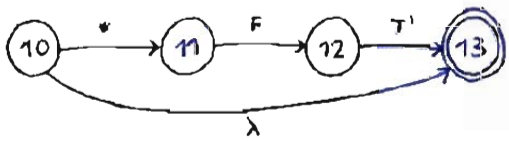


→ Representa una condición IF:
- escojo "+"
- o escojo "λ".

T:

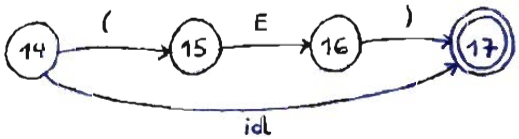
Estos diagramas IF se pueden simplificar.

T':

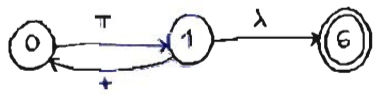


→ Representa una condición IF:
- escojo "*"
- o escojo "λ".

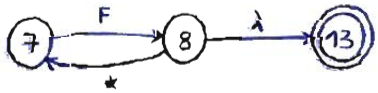
F:



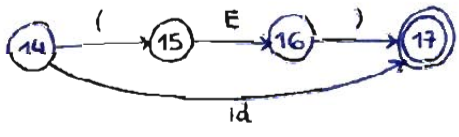
Uniendo E con E' ⇒ E:



Uniendo T con T' ⇒ T:

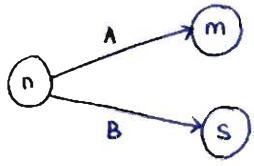


F queda igual ⇒ F:



* ¿CÓMO ELEGIR EL CAMINO?

Dada la siguiente situación en un diagrama de transición:



Tengo que saber qué camino elegir y sin equivocarme. ¿Cómo lo decido? Lo único que conozco es la cadena de entrada, en concreto el siguiente token a analizar.

Ejemplo: w = abcad → Sabe que el elemento que tiene que encontrar es "a".

Para decidir qué camino elegir se utiliza la función FIRST de los símbolos no terminales: si el símbolo terminal a la entrada pertenece al FIRST de un determinado símbolo no terminal, tengo que ir por ese camino.

La pregunta que debo contestar es:

¿ a ∈ FIRST(A) ?
¿ a ∈ FIRST(B) ?

El único problema es que "a" pertenezca a ambos FIRST, en cuyo caso tendría una ambigüedad que sólo podría resolver modificando la gramática.

Para saber si una regla tipo A → λ es la que se debe utilizar se debe mirar si el caracter a la entrada pertenece al FOLLOW(A). En caso de que sí pertenezca, se podrá utilizar dicha regla.

Ejemplo: Ahora vamos a construir los procedimientos correspondientes a los diagramas de transición de la página anterior.

```
PROCEDURE E ;
  BEGIN
      T;
      IF siguiente_token = '+' THEN
         BEGIN
            siguiente_token = scan();   -- Avanza en la lectura de la cadena de entrada.
            E;
         END
      ELSE
         BEGIN
         END
  END
```

} Corresponde a la rama λ → No hay que hacer nada, así que esta rama sobra, se puede omitir.

La rama λ irá siempre en el ELSE.

```
PROCEDURE T ;
  BEGIN
      F;
      IF siguiente_token = '*' THEN
         BEGIN
            siguiente_token = scan();
            T;
         END
  END
```

(No necesito la rama del else)

```
PROCEDURE F;
  BEGIN
      IF siguiente_token = '(' THEN
         BEGIN
            siguiente_token = scan();
            E;
            IF siguiente_token = ')' THEN
               BEGIN
                  siguiente_token = scan();
               END
```

→

```
        ELSE IF  siguiente_token = 'id' THEN
                 BEGIN
                     siguiente_token = scan();
                 END
        ELSE
                 ERROR (símbolo incorrecto);
END
```

Además, necesito un procedimiento principal que se encargue de obtener el primer token y ejecutar el axioma:

```
PROCEDURE  Comp-Desc-Rec;
  BEGIN
    siguiente_token = scan();
    E;
  END
```

→ También habría que añadir al final del procedimiento E la finalización con éxito y reconocer el símbolo especial $ (fin de la cadena).

Ejemplo:

Reconocer la cadena → id + id * id

Empiezo a ejecutar por E, axioma de la gramática. → E

T

F → id

+

E

T

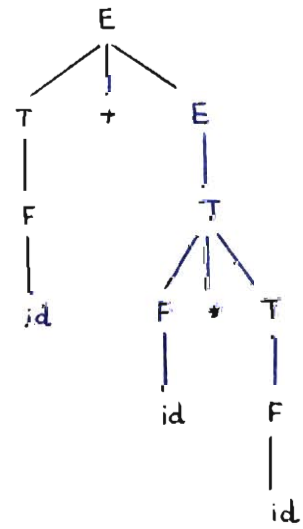
F → id

*

T

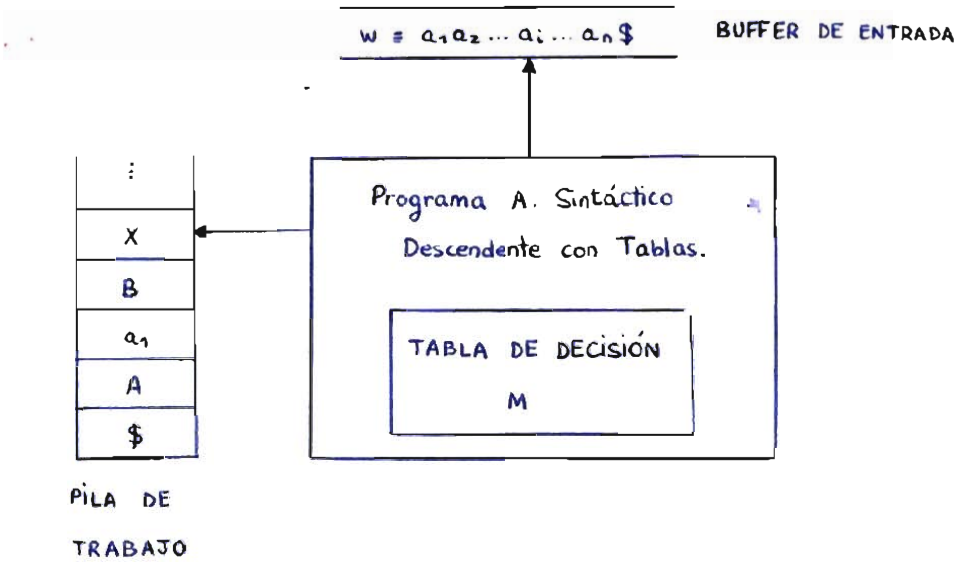
F → id

⟹



2. ANÁLISIS DESCENDENTE PREDICTIVO NO RECURSIVO = A.D. POR TABLAS = LL.

Esquema de trabajo del analizador:



BUFFER DE ENTRADA → El programa es genérico, válido para cualquier gramática no recursiva por la izquierda.

PILA DE TRABAJO → La pila se inicializa con el axioma. En ella puede haber símbolos terminales y no terminales.

TABLA DE DECISIÓN → La tabla del análisis descendente es bidimensional y en ella se toman las decisiones en función de un símbolo no terminal y de un símbolo terminal: M(A, a)

* **SITUACIONES POSIBLES:**

El programa puede tomar 2 decisiones y lo hace en función de la cabeza de la pila y el elemento siguiente de la cadena de entrada:

① Si X = a_i ⟹ Quitar X de la pila. Leer siguiente token: a_{i+1} } Validar el token (símbolo)
(X: pila, a_i: entrada)

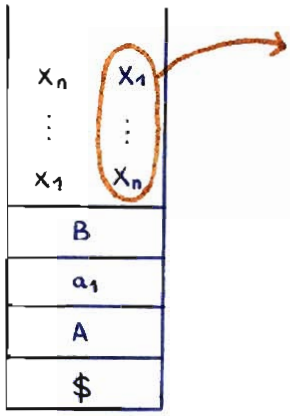
CASO ESPECIAL: Si X = a_i = $ ⟹ Aceptar la cadena. Fin.

② Si $X \in N \Rightarrow$ Consultar la Tabla de Decisión:

- Si $M(X, a_i) = \emptyset$ (casilla vacía) $\Rightarrow$ Error.
- Si $M(X, a_i) = X \to X_1 \dots X_n$ (regla de la gramática) $\Rightarrow$
  $\Rightarrow$ Quitar X de la pila.
  Meter en la pila la parte derecha de la producción.

} Derivar por esa regla.

¿Cómo introducirlo en la pila?



Esta es la forma correcta: dejando X_1 en la cima de la pila, pues lo primero que quiero derivar es X_1 (estamos en derivación por la izquierda).

ANALIZADOR SINTÁCTICO DESCENDENTE LL

Algoritmo:

Entrada: Cadena de entrada: $w=a_1, a_2, \dots a_n$
Tabla de decisión M
Salida: Si $w \in L(G)$ se obtiene el análisis a izquierdas para w; en otro caso, error
Procedimiento:

1. Pila de trabajo S inicial: $
   Buffer de entrada inicial: $w$$
2. siguiente_token = a_1 (Primer elemento de w)
   REPETIR
   X = Cima de la pila
   Si X ∈ T ó {$} entonces
   Si X = siguiente_token
   Quitar X de la pila
   Avanzar siguiente_token
   } Validar token
   Si no
   Error
   Si no // X ∈ N
   Si M(X, siguiente_token) = $X \to X_1 \dots X_n$ // Consulta tabla de decisión.
   Quitar X de la pila
   Introducir $X_1 \dots X_n$ en la pila
   } Derivar por esa regla
   Si no // casilla vacía.
   Error
   HASTA X = $
   Aceptar cadena (si X = $)

Ejemplo:

Dada la gramática:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \lambda$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \lambda$
$F \rightarrow (E) \mid id$

(Vemos que la gramática no tiene recursividad).

Dada la tabla del analizador descendente (tabla de decisión):

M	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \lambda$	$E' \rightarrow \lambda$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \lambda$	$T' \rightarrow *FT'$		$T' \rightarrow \lambda$	$T' \rightarrow \lambda$
F	$F \rightarrow id$			$F \rightarrow (E)$		

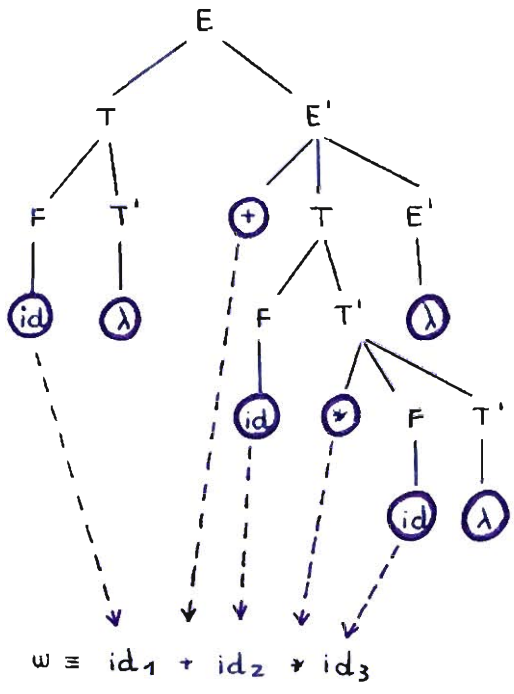
No terminales

Terminales + $

Ya veremos cómo se construye esta tabla.

Comprobar si la siguiente cadena pertenece al lenguaje generado por dicha gramática:

$w \equiv id_1 + id_2 * id_3$



Algoritmo:

PILA	SIG_TOKEN	REGLA
$E	id_1	$E \to TE'$
$E'T	id_1	$T \to FT'$
$E'T'F	id_1	$F \to id$
$E'T'id	id_1	–
$E'T'	+	$T' \to \lambda$
$E'	+	$E' \to +TE'$
$E'T+	+	–
$E'T	id_2	$T \to FT'$
$E'T'F	id_2	$F \to id$
$E'T'id	id_2	–
$E'T'	*	$T' \to *FT'$
$E'T'F*	*	–
$E'T'F	id_3	$F \to id$
$E'T'id	id_3	–
$E'T'	$	$T' \to \lambda$
$E'	$	$E' \to \lambda$
$	$	Aceptar

Ascendente Vs. Descendente:

- Reducir es ahora expandir.
- Desplazar es ahora consumir el terminal.

2.1. CONSTRUCCIÓN DE LA TABLA DE DECISIÓN.

Entrada: Gramática no recursiva por la izquierda, factorizada, condición LL.

Salida: Tabla de decisión $M(A, a)$ „ $A \in N$, $a \in T \cup \{\$\}$

Método: * Para cada producción $A \to \alpha \in$ Gramática „ $\alpha \in (N \cup T)^*$

1) $\forall t \in First(\alpha) \Rightarrow M(A,t) = A \to \alpha$

2) Si $\lambda \in First(\alpha) \Rightarrow \forall t \in Follow(A) \;/\; t \in T \cup \{\$\}$
$M(A,t) = A \to \alpha$

* Casillas vacías $\Rightarrow$ Error.

⚠ En la tabla no puede haber ni filas ni columnas completamente en blanco. Una columna en blanco significa que el terminal nunca se obtiene, por lo que la gramática estaría mal construida o el terminal sería innecesario. Una fila en blanco significa que el símbolo no terminal no se puede derivar.

Ejemplo: Regla 1.

$B \to aC$
$B \to bC$
$C \to \lambda$

Para la regla $B \to bC$ tenemos:

$First(bC) = \{b\}$

$\Downarrow$

$M(B, b) = B \to bC$

Ejemplo: Regla 2.

$C \to D$
$C \to bD$
$D \to \lambda$
$D \to cD$
$D \to d$

$C \overset{*}{\to} \lambda$

Para la regla $C \to D$ tenemos:

$\lambda \in First(D)$

$\Downarrow$

$c \in Follow(C)$

$\Downarrow$

$M(C, c) = C \to D$

Ejemplo: Construcción de la tabla de decisión.

$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$

Modificamos la gramática para que cumpla las condiciones necesarias.

$E \to TE'$
$E' \to +TE' \mid \lambda$
$T \to FT'$
$T' \to *FT' \mid \lambda$
$F \to (E) \mid id$

Gramática válida

$T = \{ id, +, *, (,) \}$
$N = \{ E, E', T, T', F \}$

* Calculamos los First y Follow:

First (E) = { (, id }
First (T) = { (, id }
First (F) = { (, id }
First (E') = { +, λ }
First (T') = { *, λ }

Follow (E) = { $,) } ← por ser E el axioma.
Follow (T) = { +, $,) }
Follow (F) = { *, +, $,) }
Follow (E') = { $,) }
Follow (T') = { +, $,) }

* Construcción de la tabla de decisión:

M	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → λ	E' → λ
T	T → FT'			T → FT'		
T'		T' → λ	T' → *FT'		T' → λ	T' → λ
F	F → id			F → (E)		

⇒ Follow de E'.

E → First (TE') = First (T) = { (, id }

E' → First (+TE') = { + }
Follow (E') = { $,) } (λ ∈ First (α))

T → First (FT') = First (F) = { (, id }

T' → First (*FT') = { * }
Follow (T') = { +, $,) } (λ ∈ First (α))

F → First ((E)) = { (}
First (id) = { id }

2.2. GRAMÁTICA LL(1)

Las gramáticas LL(1) son las que se pueden utilizar para construir los analizadores predictivos.

Las propiedades de estas gramáticas son las siguientes:

1.- No ambigua (sólo un árbol de derivación a izquierdas nos lleva a la cadena a analizar).

2.- No recursiva por la izquierda (eliminar siempre esta recursividad para no entrar en un bucle infinito).

3.- Factorizada por la izquierda (sólo una regla que empiece de la misma forma).

4.- $\forall A \in N$, si $\exists A \to \alpha$ y $\exists A \to \beta$ " $\alpha \neq \beta \Rightarrow$

$\Rightarrow First(\alpha) \cap First(\beta) = \phi$

($\alpha \overset{*}{\Rightarrow} \lambda$)

Además, si $\lambda \in First(\alpha) \Rightarrow$

$\Rightarrow Follow(A) \cap First(\beta) = \phi$ → Equivalente para más de 2 reglas.

Dada $A \to \alpha_1$, $A \to \alpha_2$, ⋮, $A \to \alpha_n$

Si $\lambda \in First(\alpha_n) \Rightarrow$

$\Rightarrow \lambda \notin First(\alpha_1), \dots, First(\alpha_{n-1}) \wedge$

$First(\alpha_1) \cap Follow(A) = \phi$

...

$First(\alpha_{n-1}) \cap Follow(A) = \phi$

Ejemplo: Dada una sentencia IF, tengo dos posibilidades:

Sent IF → IF Cond THEN Sent ELSE Sent
Sent IF → IF Cond THEN Sent

$A \to \pm B$
$A \to \pm D$
} Otro caso

→

Para el mismo símbolo no terminal tengo 2 derivaciones (reglas) que comienzan con el mismo símbolo ⇒ No está factorizada la gramática ⇒ ⇒ La gramática no es LL(1):

$$First(+B) \cap First(+D) = \{+\} \neq \phi$$

En consecuencia, cuando tenga el símbolo "A" en la cima de la pila y el símbolo terminal "+" como siguiente token a la entrada, no sabré qué regla aplicar.

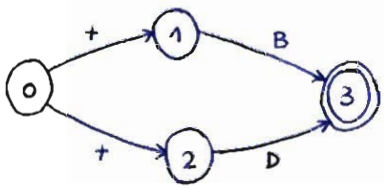
¿Diferencia entre el método predictivo recursivo y el método predictivo no recursivo?

Con el NO RECURSIVO tengo que cambiar la gramática, necesito que sea una gramática LL(1).
Con el RECURSIVO es conveniente cambiar la gramática porque me quito problemas, pero se puede no cambiar y hacer un apaño.

Método Predictivo Recursivo:

* Diagrama de transición:

A:

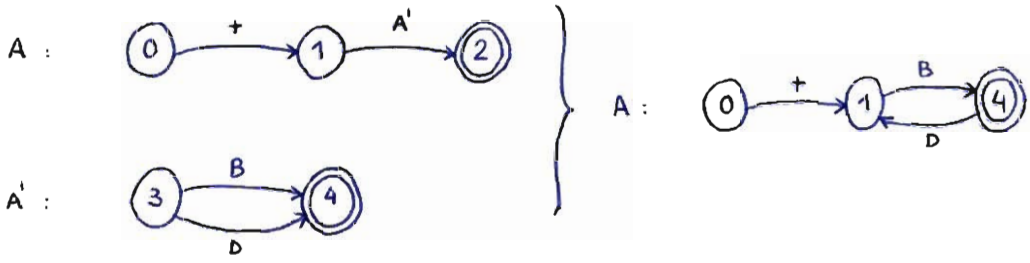


* Procedimiento:

```
PROCEDURE  A;
BEGIN
    IF  siguiente_token = '+'  THEN
        siguiente_token = scan();          -- Leer otro caracter. No decido por '+' porque
    IF  siguiente_token ∈ First(B)  THEN      es un factor común.
        B;
    ELSE
        D;
END A;
```

El apaño consiste en retrasar la decisión de qué procedimiento ejecutar: valido el token '+' y tomo la decisión en función del siguiente token. Así, en lugar de cambiar la gramática, altero el procedimiento para sacar factor común.

Si transformamos la gramática: $A \rightarrow + A'$; $A' \rightarrow B \mid D$



Ejercicio: Analizador Sintáctico Descendente Predictivo Recursivo.

Gramática: Tipo → Simple | Array [Simple] of Tipo
Simple → integer | char | num .. num

T = { Array, [,] , of , integer , char , num .. num }

N = { Tipo , Simple }

```
PROCEDURE Valida ( t : token)
BEGIN
    IF siguiente_token = t THEN
        siguiente_token = scan ();
    ELSE
        Error ;
    END IF
END
```

Vamos a calcular los procedimientos del Analizador Sintáctico Descendente Predictivo Recursivo (es uno por cada símbolo no terminal).

→

```
PROCEDURE  Tipo
BEGIN                                  → Cuando el siguiente símbolo es un no terminal.
    IF  siguiente_token ∈ First(Simple) THEN
        Simple;
    ELSE IF  siguiente_token = 'Array' THEN
        BEGIN                          → Cuando el siguiente símbolo es un terminal.
            Valida ('Array');
            Valida ('[');
            Simple;
            Valida (']');
            Valida ('of');
            Tipo;
        END
    END IF
END
```

```
PROCEDURE  Simple
BEGIN
    IF  siguiente_token = 'integer' THEN
        Valida ('integer');
    ELSE IF  siguiente_token = 'char' THEN
        Valida ('char');
    ELSE
        Valida ('num..num');
    END IF
END
```

ANÁLISIS SINT. DESCENDENTE PREDICTIVO RECURSIVO ⇒ Diagramas de transición y procedimientos.

ANÁLISIS SINT. DESCENDENTE PREDICTIVO NO RECURSIVO (POR TABLAS O LL) ⇒ Tabla de decisión y algoritmo.

Ejercicio: Analizador Sintáctico Descendente Predictivo Recursivo.

Dada la gramática en notación EBNF:

Bloque → begin Sent { ; Sent } end
Sent_if → if Cond then Sent [else Sent]
Cond → id > id
Sent_asig → id := id { + id }
Sent → Sent_if | Sent_asig | Bloque | λ

{ } Repetición ϕ o más veces.
[ ] Repetición ϕ o 1 vez.

T = { begin, ; , end , if , then , else , id , > , := , + }

N = { Bloque, Sent_if , Cond , Sent_asig , Sent }

Los [] y {} no son símbolos de la gramática, son símbolos de la notación.

Calculamos los procedimientos:

```
PROCEDURE Bloque ;
BEGIN
      Valida ('begin');
      Sent;
      WHILE siguiente_token = ';' DO
      BEGIN
            Valida (';');
            Sent;
      END
      Valida ('end');
END
```

Para la repetición de ϕ o más veces : {} en notación EBNF.

```
PROCEDURE Sent_if ;
BEGIN
      Valida ('if');
      Cond;
      Valida ('then');
      Sent;
      IF siguiente_token = 'else' THEN
      BEGIN
            Valida ('else');
            Sent;
      END
END
```

Para la repetición de ϕ o 1 vez: [] en notación EBNF.

```
PROCEDURE  Sent ;
BEGIN
     IF  siguiente_token = 'if' THEN
         Sent_if ;
     ELSE IF  siguiente_token = 'id' THEN
         Sent_asig;
     ELSE IF siguiente_token = 'begin' THEN
         Bloque ;
     END IF ;
END

PROCEDURE  Cond;
BEGIN
     Valida ('id');
     Valida ('>');
     Valida ('id');
END

PROCEDURE  Sent_asig;
BEGIN
     Valida ('id');
     Valida (':=');
     Valida ('id');
     WHILE  siguiente_token = '+' DO
     BEGIN
          Valida ('+');
          Valida ('id');
     END
END
```

Hemos trabajado con la gramática en notación EBNF. Si la hubiéramos cambiado para pasarla al formato normal, sería:

Bloque → begin Sent {; Sent} end

{ Bloque → begin Sent A
  A → end | ; Sent A

Sent_if → if Cond then Sent [else Sent]

{ Sent_if → if Cond then Sent B
  B → λ | else Sent

Ejercicio: Construcción de la tabla de decisión.

F → (id A)
A → P O R
O → P O | null
R → rest O | λ
P → (id O)

T = { id , (,) , null , rest }

N = { F , A , O , R , P }

* Calculamos los First y Follow:

First (F) = { (}
First (A) = { (}
First (O) = { (, null }
First (R) = { rest , λ }
First (P) = { (}

→ F es el axioma (único símbolo no terminal que no aparece en la parte derecha de las producciones).

Follow (F) = { $ }
Follow (A) = {) }
Follow (O) = { rest ,) }
Follow (R) = {) }
Follow (P) = { (, null }

* Construcción de la tabla de decisión:

M	id	(	)	null	rest	$
F		F → (id A)				
A		A → P O R				
O		O → P O		O → null		
R			R → λ		R → rest O	
P		P → (id O)				

F → First ((id A)) = { (}

A → First (P O R) = { (}

O → First (P O) = { (}
First (null) = { null }

R → First (rest O) = { rest }
First (λ) = λ ⇒ Follow (R) = {) }

P → First ((id O)) = { (}

→ Quedan columnas en blanco ⇒ Símbolos terminales innecesarios o gramática mal construida.

Si a la gramática anterior le añadimos la producción: $P \to \lambda$, tenemos los siguientes cambios:

First (A) = { (, null }

First (P) = { (, λ }

M	id	(	)	null	rest	$
F		F → (id A)				
A		A → P O R		A → POR		
O		O → PO		O → PO O → null		
R			R → λ		R → rest O	
P		P → (id O) P → λ			P → λ	

A → First (P O R) = { (, null }

O → First (P O) = { (, null }

First (null) = { null }

En 2 casillas hay 2 reglas ⇒ Gramática inválida.

Ejercicio: Comprobación gramáticas LL(1).

F → (id A)

A → P O R

O → P O | null

R → rest O | λ

P → (id O) | λ

* Miramos O:

O → P O

O → null

First (P O) ∩ First (null) = { (, null } ∩ { null } ≠ ∅

No es ∅ ⇒ No se cumple la condición LL.

Ya sabemos que la gramática no es LL, pero comprobamos los demás pares de reglas simplemente por practicar.

* Miramos R:

$R \to rest\ 0$
$R \to \lambda$

$First(rest\ 0) \cap First(\lambda) = \{rest\} \cap \{\lambda\} = \emptyset$

$Follow(R) \cap First(rest\ 0) = \{)\} \cap \{rest\} = \emptyset$

⇓

R no da problemas.

* Miramos P:

$P \to (\ id\ 0\ )$
$P \to \lambda$

$First((id\ 0)) \cap First(\lambda) = \{(\} \cap \{\lambda\} = \emptyset$

$Follow(P) \cap First((id\ 0)) = \{(, null\} \cap \{(\} \neq \emptyset$

⇓

Las reglas de P dan problemas.

Ejercicio: Comprobación gramáticas LL(1).

$S \to 0AA0 \mid 1A1 \mid \lambda$
$A \to B \mid C$
$B \to 01 \mid 10$
$C \to 00 \mid 11$

- Gramática no recursiva por la izquierda.
- Gramática factorizada.
- Comprobar condición LL:

$First(S) = \{0, 1, \lambda\}$ $\quad Follow(S) = \{\$\}$
$First(A) = \{0, 1\}$ $\quad Follow(A) = \{0, 1\}$
$First(B) = \{0, 1\}$ $\quad Follow(B) = \{0, 1\}$
$First(C) = \{0, 1\}$ $\quad Follow(C) = \{0, 1\}$

→

* Comprobamos S:

$S \to 0\,A\,A\,0$
$S \to 1\,A\,1$
$S \to \lambda$

$First(0AA0) \cap First(1A1) = \{0\} \cap \{1\} = \emptyset$
$First(0AA0) \cap First(\lambda) = \{0\} \cap \{\lambda\} = \emptyset$
$First(1A1) \cap First(\lambda) = \{1\} \cap \{\lambda\} = \emptyset$

$First(0AA0) \cap Follow(S) = \{0\} \cap \{\$\} = \emptyset$
$First(1A1) \cap Follow(S) = \{1\} \cap \{\$\} = \emptyset$

⇓

S no da problemas.

* Comprobamos A:

$A \to B$
$A \to C$

$First(B) \cap First(C) = \{0,1\} \cap \{0,1\} \neq \emptyset$

⇓

No se cumple la condición LL.

* Comprobamos B:

$B \to 01$
$B \to 10$

$First(01) \cap First(10) = \{0\} \cap \{1\} = \emptyset$

⇓

B no da problemas.

* Comprobamos C:

$C \to 00$
$C \to 11$

$First(00) \cap First(11) = \{0\} \cap \{1\} = \emptyset$

⇓

C no da problemas.

Ejercicio:

$S \rightarrow A\,B\,C$

$A \rightarrow \lambda \mid A\,*$

$B \rightarrow C\,B \mid +$

$C \rightarrow -$

a) Comprobar condiciones LL.

- Es recursiva por la izquierda. Hay que arreglarla:

$S \rightarrow A\,B\,C$

$A \rightarrow \lambda \mid *\,A$ → Cambiando el orden de estos símbolos ya no es recursiva por la izquierda.

$B \rightarrow C\,B \mid +$

$C \rightarrow -$

- La gramática ya está factorizada.

- Comprobar condición LL:

$First(S) = \{*, +, -\}$ $\quad Follow(S) = \{\$\}$

$First(A) = \{\lambda, *\}$ $\quad Follow(A) = \{+, -\}$

$First(B) = \{+, -\}$ $\quad Follow(B) = \{-\}$

$First(C) = \{-\}$ $\quad Follow(C) = \{+, -, \$\}$

* Miramos A:

$A \rightarrow *\,A$
$A \rightarrow \lambda$

$First(*A) \cap First(\lambda) = \{*\} \cap \{\lambda\} = \emptyset$

$Follow(A) \cap First(*A) = \{+, -\} \cap \{\lambda, *\} = \emptyset$

⇓

Las reglas de A no dan problemas.

* Miramos B:

$B \rightarrow C\,B$
$B \rightarrow +$

$First(CB) \cap First(+) = \{-\} \cap \{+\} = \emptyset$

⇓

Las reglas de B no dan problemas.

b) Calcular los procedimientos (código) del A. S. Descendente Predictivo Recursivo.

```
                    // Valida
PROCEDURE  Empareja  (t : token)
BEGIN
    IF  siguiente_token = t  THEN
        siguiente_token = scan ();      -- Leer siguiente token.
    ELSE
        error (t);
    END IF;
END
```

```
PROCEDURE   Desc_Rec;              -- Procedimiento principal.
BEGIN
    siguiente_token = scan ();
    S;
END
```

```
PROCEDURE  S;
BEGIN
    A;
    B;
    C;
END
```

```
PROCEDURE  A;
BEGIN
   IF  siguiente_token = '*'  THEN
       Empareja (*);
       A;
   END
```

Rama para $A \to \lambda$.

En el ELSE se podría comprobar si el token siguiente está en el follow (A), pero mejor ya lo comprobamos en el procedimiento B. No adelantamos la detección de errores.

```
PROCEDURE  B ;
BEGIN
    IF  siguiente_token ='-' THEN
        C ;
        B ;
    ELSE
        Empareja (+);
    END IF
END
```

```
PROCEDURE  C;
BEGIN
    Empareja (-);
END
```

c) Construcción tabla de decisión (A.S. Descendente Predictivo no Recursivo = LL).

$T = \{ *, +, - \}$

$N = \{ S, A, B, C \}$

M	*	+	-	$
S	S → ABC	S → ABC	S → ABC	
A	A → *A	A → λ	A → λ	
B		B → +	B → CB	
C			C → -	

S → First (ABC) = { *, +, - }

A → First (* A) = { * }

First (λ) = λ ⇒ Follow (A) = { +, - }

B → First (CB) = { - }

First (+) = { + }

C → First (-) = { - }

d) Justificar si la gramática es de precedencia de operador. → Pág. 35

No, la gramática no cumple las reglas necesarias para ser de precedencia de operador: hay reglas con más de un no terminal seguido y hay reglas λ.

Se cambia la gramática:

$S \to A * B$
$A \to *$
$B \to B + C - | +$
$C \to -$

Ahora para ver si es válida hay que construir la tabla.

1)

Head (S) = {*}	Tail (S) = {*, +, -}
Head (A) = {*}	Tail (A) = {*}
Head (B) = {+}	Tail (B) = {+, -}
Head (C) = {-}	Tail (C) = {-}

2)

* $S \to A * B$
  N T N
  $X_1\ X_2\ X_3$

  $i = 1 \to$ Tail (A) $\gtrdot$ * $\Rightarrow$ {*} $\gtrdot$ *
  $i = 2 \to$ * $\lessdot$ Head (B) $\Rightarrow$ * $\lessdot$ {+}

* $A \to *$ (No hago nada)

* $B \to B + C -$
  N T N T
  $X_1 X_2 X_3 X_4$

  $i = 1 \to$ Tail (B) $\gtrdot$ + $\Rightarrow$ {+, -} $\gtrdot$ +
  $i = 2 \to$ + $\doteq$ -, + $\lessdot$ Head (C) $\Rightarrow$ + $\lessdot$ -
  $i = 3 \to$ Tail (C) $\gtrdot$ - $\Rightarrow$ {-} $\gtrdot$ -

* $B \to +$ (No hago nada)

* $C \to -$ (No hago nada)

3) $\$ \lessdot Head(S) \Rightarrow \$ \lessdot \{*\}$

$Tail(S) \gtrdot \$ \Rightarrow \{*, +, -\} \gtrdot \$$

T.R.	*	+	-	$
*	$\gtrdot$	$\lessdot$		$\gtrdot$
+		$\gtrdot$	$\doteq$ $\lessdot$	$\gtrdot$
-		$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$			

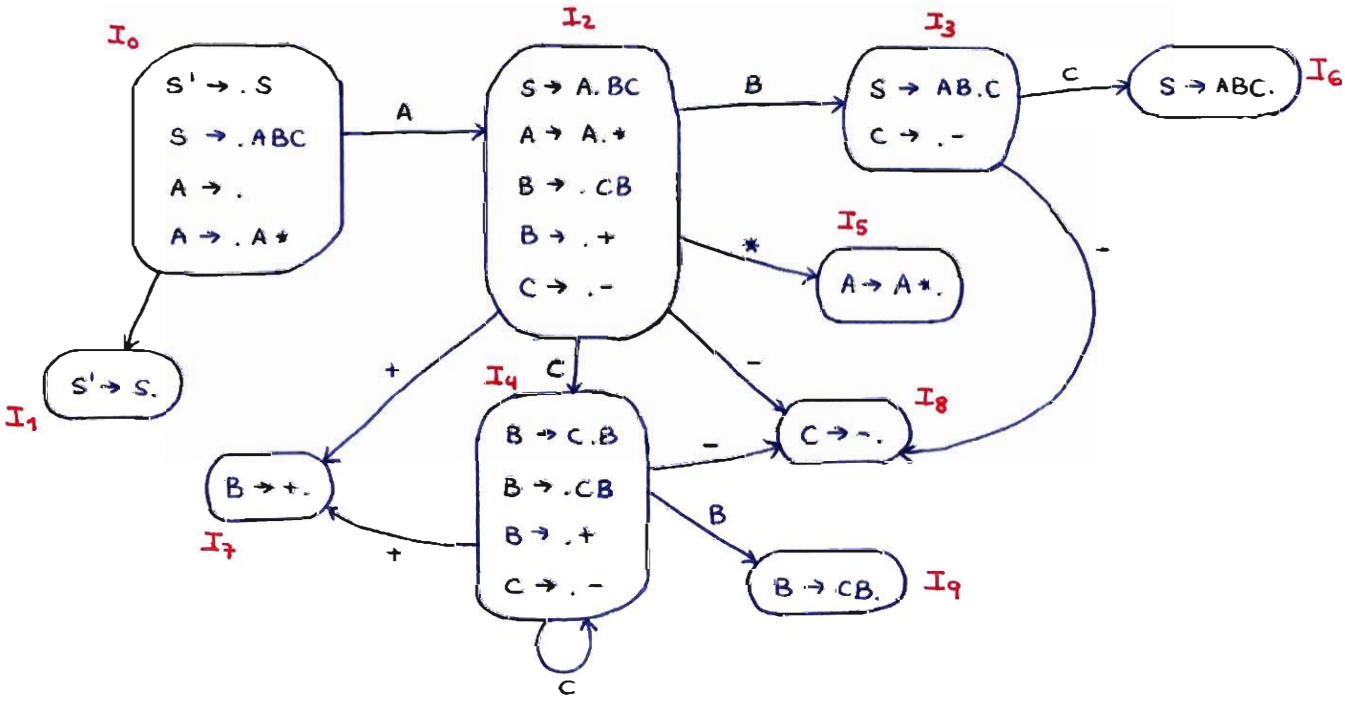
El método no es de precedencia de operador.

e) Análisis ascendente LR. Autómata.

$S \to ABC$
$A \to \lambda \mid A*$
$B \to CB \mid +$
$C \to -$
$S' \to S$

} Gramática aumentada.

En lugar de hacer todo el proceso, hacemos el autómata directamente.



No hay conflictos Reducción - Reducción.

No hay conflictos Reducción - Desplazamiento.

EXAMEN FEBRERO 2008.

La Biblioteca de la Facultad de Informática quiere crear una aplicación para almacenar en su Base de Datos toda la información sobre los Trabajos Fin de Carrera que tiene almacenados en un fichero XML. El formato de los datos en este fichero es el siguiente:

```
<tfc>
   <autor>Luis Gómez y Villalonga de Pi</autor>
   <fecha_nac>18/05/85</fecha_nac>
   <titulo_tfc>Diseño 3o de un Reconocedor de TFC's </titulo_tfc>
   <tutor> Mariam Heras</tutor>
   <nota>sobresaliente 9</nota>
</tfc>
```

Deben tenerse en cuenta las siguientes características del formato del fichero:

- El título admite todo tipo de caracteres excepto el signo menor ("<"). El título de un TFC siempre empieza por una letra.
- La nota numérica puede ir de 0 a 10 y puede estar escrita con un decimal.
- La nota no numérica será una de las palabras: {"matrícula", "sobresaliente", "notable", "aprobado", "suspenso"}.
- En un TFC puede aparecer solo un tipo de nota o ambas y, en este caso, en cualquier orden.
- Las etiquetas XML son de dos tipos: de apertura y de cierre. Cada etiqueta contiene una palabra encerrada entre "<" y ">"; las de cierre llevan, además, el símbolo "/".
- Existe una tabla que contiene todas las palabras clave de las etiquetas XML válidas: {"tfc", "autor", "fecha_nac"...}. Las palabras sólo contienen letras y subrayados.
- Las fechas deben tener dos dígitos para el día, dos para el mes y dos para el año, y debe ser una fecha correcta. En la Base de Datos se almacenará como cadena.

Se pide construir un Analizador Léxico para este lenguaje (tokens, gramática, autómata finito determinista y acciones semánticas).

Tokens:

(etiq_inic, pos)
(etiq_final, pos)
(nota_num, valor)
(nota_simb, código)
(fecha, cadena)
(cad, cadena)

Gramática:

$c = T - \{ \ell, d, < \}$

$S \rightarrow < E \mid \ell C \mid dN \mid del\ S$

$E \rightarrow \ell A \mid / B$

$A \rightarrow \ell A \mid \_ A \mid >$

$B \rightarrow \ell B \mid \_ B \mid >$

$C \rightarrow \ell C \mid dC \mid cC \mid \lambda$

$N \rightarrow dN' \mid .N'' \mid \lambda$

$N' \rightarrow /M \mid .N'' \mid \lambda$

$N'' \rightarrow d$

$M \rightarrow dM'$

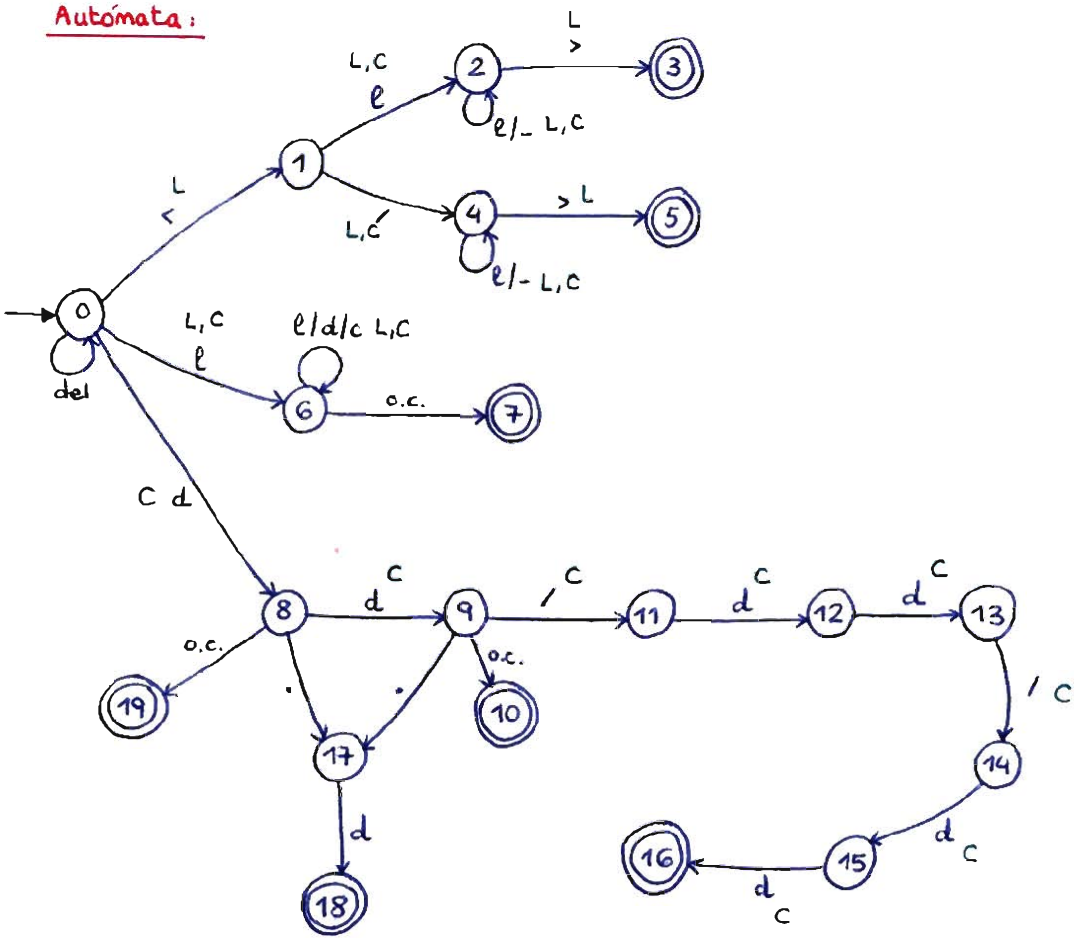
$M' \rightarrow dM''$

$M'' \rightarrow /Y$

$Y \rightarrow dY'$

$Y' \rightarrow d$

Autómata:



Acciones semánticas:

- Lee en todas las transiciones excepto en las de o.c. (otro caracter).

- C = Concatenar.

- Para las fechas:

```
0-8 → Num := d;
8-9 → Num := Num * 10 + d;
11-12 → Mes := d;
12-13 → Mes := Mes * 10 + d;
14-15 → Año := d;
15-16 → Año := Año * 10 + d;
        if not Es_Correcta (Num, Mes, Año) then
           error;
        else
           Gen_token (fecha, "Num +/+ Mes +/+ Año");
```

- Para las notas:

```
9-10 → if (Num > 10) then
8-19 ↗     error ("Nota > 10");
        else
           Gen_token (nota_num, Num);
17-18 → Num := Num + d / 10;
        if (Num ≤ 10) then
           Gen_token (nota_num, Num);
        else
           error ("Nota > 10");
```

- Toda transición no prevista → Error.

- Para las etiquetas:

```
2-3 → p := Busca (pal);
4-5 ↗ if (p = null) then
         error ("Etiqueta no válida");
      else
         Gen_token (etiq_inic, p); / Gen_token (etiq_final, p);
```

2-3 4-5

- Para las cadenas:

```
6-7 → p := Busca (pal);
      if (p ≠ null) then Gen_token (nota_simb, p);
      else Gen_token (cad, pal);
```